TP8 : Tic-tac-toe

Objectifs du TP :

- Faire quelques révisions sur la structure de dictionnaire.
- Revoir la notion de linéarisation d'un tableau et de sérialisation.
- Calculer les attracteurs d'un jeu et en déduire un algorithme permettant à un ordinateur de jouer au jeu concerné en suivant une stratégie optimale.

Partie 0 *Préliminaires*

Ce sujet fournit un module dicos qui servira dans la partie 2 ainsi qu'un makefile permettant de compiler le source TP7.c en y liant dicos. Commencez par copier tous ces fichiers dans votre répertoire personnel.

Dans ce TP, on considère une version généralisée du tic-tac-toe (plus connu sous le nom de morpion) : le principe reste le même mais le jeu, noté $T(n,k)$, se joue sur une grille carrée de taille n et l'objectif est d'être le premier à obtenir un alignement de k symboles identiques en ligne, colonne ou diagonale.

Pour encoder une configuration du jeu $T(n,k)$ en C, on utilise la structure suivante :

```
struct ttt {
    int k;
    int n;
    int* grille;
};
typedef struct ttt ttt;
```

De manière transparente, si jeu est un pointeur sur un objet de type ttt qui représente une configuration du jeu $T(n,k)$, jeu->n correspond à n et jeu->k correspond à k. De plus, jeu->grille est un tableau unidimentionnel de taille n^2 linéarisant la grille associée à la configuration considérée avec les conventions suivantes. Une case ne peut contenir que $0, 1$ ou 2 : 0 correspond aux cases vides, 1 aux cases occupées par les symboles du joueur 1 et 2 aux cases occupées par les symboles du joueur 2. Par exemple :

1		1
2	1	
2		

est implémentée par le tableau $[1,0,1,2,1,0,2,0,0]$.

1. Ecrire une fonction int numero_case(ttt* jeu, int l, int c) qui indique quel est le numéro de la case de jeu->grille se trouvant en ligne l et en colonne c (les numérotations commençant par convention à zéro). Par exemple, sur une grille 3×3, la case en ligne 1 et colonne 2 porte le numéro 5.
2. Ecrire une fonction réciproque de la précédente void numeros_lc(ttt* jeu, int position, int* l, int* c) stockant dans l (respectivement c) la ligne (respectivement la colonne) à laquelle se trouve la case numéro position dans jeu->grille.

Ces fonctions permettront de faire les traductions entre la manipulation de la grille par un humain (en deux dimensions) et la machine (en une dimension).

3. Ecrire une fonction ttt* initialiser_jeu(int n, int k) qui renvoit un pointeur sur la configuration initiale du jeu $T(n,k)$. Toutes les cases de la grille y sont vides.
4. Ecrire une fonction void liberer_jeu(ttt* jeu) permettant de libérer la mémoire allouée sur le tas lors de la création d'un objet de type ttt* via initialiser_jeu.

5. Ecrire une fonction int* nb_symboles(ttt* jeu) renvoyant un tableau de taille 3 et contenant en case i le nombre de symboles égaux à i dans la grille de jeu. A l'aide d'un assert, on s'assurera que ladite grille ne contient pas de symbole autre que $0, 1, 2$.
6. En déduire une fonction int joueur_courant(ttt* jeu) renvoyant le numéro du joueur qui doit jouer le prochain coup (même si la partie est finie). On se fixe la convention selon laquelle le joueur qui commence est toujours le joueur 1.
7. En déduire également un fonction bool grille_remplie(ttt* jeu) indiquant si la grille est remplie.
8. Ecrire une fonction void jouer_coup(ttt* jeu, int l, int c) qui modifie la configuration jeu de manière à simuler le fait que le joueur courant joue un coup en ligne l et colonne c. On supposera sans le vérifier que le coup est licite, c'est-à-dire que la case en position (l, c) dans la grille de jeu existe et est vide.

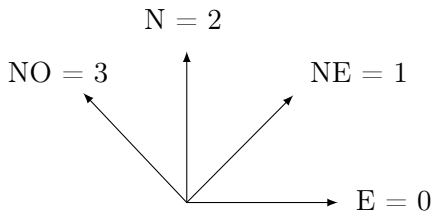
Pour la suite du TP, on suit le fil directeur suivant :

- Dans la partie 1, on détermine les configurations finales et gagnantes pour chacun des joueurs.
- Cela nous permettra dans la partie 2 de calculer les attracteurs de ce jeu et d'en déduire une stratégie de jeu optimale pour chacun des deux joueurs.
- Finalement, dans la partie 3, on implémente une interface rudimentaire permettant de jouer au tic-tac-toe généralisé contre un ordinateur qui joue selon la stratégie optimale précédente.

Partie 1 *Détermination des positions gagnantes*

Pour déterminer si une configuration est finale et gagnante pour un joueur i, on procède de la façon suivante : pour chaque case c, pour chaque direction d parmi les 8 possibles, on regarde si les k cases à partir de c dans la direction d contiennent le symbole du joueur i.

9. Pourquoi est-il suffisant de ne considérer que les quatre directions Est, Nord-Est, Nord et Nord-Ouest pour vérifier si un joueur est gagnant ? Dans la suite, on attribue un code numérique à chacune de ces directions avec les conventions ci-dessous :



10. Ecrire une fonction bool alignement(ttt* jeu, int pos, int d, int j) qui renvoit true si et seulement si le joueur j a un alignement de ses symboles à partir de la case pos dans la direction d.
11. En déduire une fonction bool est_final(ttt* jeu, int j) qui renvoit true si et seulement si la configuration en entrée est finale et gagnante pour le joueur j.

Partie 2 *Calcul d'une stratégie optimale*

L'objectif de cette partie est de déterminer pour chaque configuration s de $T(n, k)$ dans quel attracteur elle se trouve. On procède de la façon suivante :

- Si s est une configuration finale, on peut immédiatement conclure. On prend comme convention qu'une partie nulle est dans l'attracteur du "joueur" 0.

- Sinon, on calcule l'ensemble S des configurations qui peuvent suivre la configuration s et on calcule récursivement pour chacune dans quel attracteur elle se trouve.
  - → Si une de ces configurations est dans l'attracteur du joueur j qui doit jouer en s, alors la configuration s est elle aussi dans l'attracteur de j.
  - → Dans le cas contraire, deux cas sont possibles. Si au moins une des configurations de S est dans l'attracteur de 0, alors s aussi, le joueur j préférant faire nul s'il ne peut pas gagner. Sinon, toutes les configurations qui suivent s sont dans l'attracteur du joueur opposé à j et donc s aussi.

Cet algorithme n'est rien d'autre que l'algorithme min-max déguisé ! Afin d'éviter de calculer plusieurs fois le numéro de l'attracteur d'une configuration lors des appels récursifs, on mémorise ces derniers une fois calculés pour la première fois à l'aide d'un dictionnaire implémenté par table de hachage. Les opérations disponibles sur ces dictionnaires sont :

- `dico* creer_dico(void)` permettant de créer un dictionnaire vide.
- `void liberer_dico(dico* d)` permettant de libérer la mémoire utilisée sur le tas par un dictionnaire.
- `bool appartient_dico_cle(dico* d, int64_t k)` indiquant si la clé `k` est présente dans le dictionnaire.
- `int valeur_associee(dico* d, int64_t k)` permettant de récupérer la valeur associée à la clé `k`.
- `void ajoute_entree(dico* d, int64_t k, int v)` ajoutant une association (clé, valeur) au dictionnaire.

12. A une grille de $T(n,k)$ correspond un entier écrit en base 3 de taille n^2. Associer à une configuration cet identifiant entier permet de la distinguer de manière unique. Ecrire une fonction `int64_t encoder_jeu(ttt* jeu)` calculant le numéro de la configuration prise en entrée.

On utilisera ces numéros comme clés du dictionnaire stockant les numéros d'attracteurs déjà calculés. Par exemple, la configuration en début d'énoncé porte le numéro 7875 et est gagnante pour 1 : une fois le calcul du numéro d'attracteur de celle-ci effectué, on devra donc trouver l'association $(7875, 1)$ dans le dictionnaire.

13. Ecrire une fonction `int attracteur(ttt* jeu, dico* d)` prenant en argument une configuration et un dictionnaire et renvoyant le numéro de l'attracteur auquel appartient la configuration. La fonction devra mémoriser le résultat dans le dictionnaire si il n'y est pas déjà avant de le renvoyer.
14. En déduire une fonction `int coup_optimal(ttt* jeu, dico* d)` qui détermine une position dans la grille à laquelle il est optimal de jouer étant donnée une configuration non finale (on ne vérifiera pas qu'elle l'est) et un dictionnaire servant à stocker les numéros des attracteurs.

Partie 3 *Ready player one ?*

15. Ecrire une fonction `void afficher_grille(ttt* jeu)` permettant de visualiser dans la console l'état de la grille d'une configuration. On indiquera les numéros des lignes et des colonnes afin de pouvoir déterminer facilement la case où l'on souhaitera jouer par la suite. Par exemple, on peut aboutir au résultat suivant :

```
  0 1 2 3
0|O|O| | |
1| |X| | |
2| | |X| |
3| | | | |
```

16. A l'aide des questions précédentes, écrire une fonction `void jouer_partie(int n, int k)` qui créé une partie de $T(n,k)$ et permet de jouer contre l'ordinateur. La fonction devra :

- Demander au joueur humain s'il veut commencer ou non,
- Demander à chaque coup du joueur humain la ligne et la colonne où il souhaite jouer,
- Afficher la grille après chaque coup, qu'il soit joué par la machine ou l'humain,
- Arrêter la partie avec un message d'erreur explicite si le joueur humain demande à jouer dans une case qui n'existe pas ou est déjà occupée,
- Arrêter la partie dès qu'un joueur a gagné ou que la grille est pleine en indiquant le joueur gagnant.

On rappelle la syntaxe de scanf qui vous sera utile pour faire interagir l'humain avec ce programme.

```
char lettre;
printf("Voulez-vous commencer ? (o/n) ");
scanf("%c", &lettre);
```

Bonus : Vous aurez sans doute remarqué que dans ce TP on recalcule fréquemment des informations qu'on pourrait déduire des calculs précédents (par exemple, quel est le joueur qui doit jouer dans une configuration donnée). Proposer et implémenter des modifications permettant d'améliorer ce point. Pourquoi n'est-ce malgré tout pas si grave de répéter ces calculs ?